Visual Code Annotations for Cyberphysical Programming

Ben Swift*, Andrew Sorensen†, Henry Gardner*, John Hosking*
*Resarch School of Computer Science, Australian National University, Australia
{ben.swift, henry.gardner, john.hosking}@anu.edu.au
†Institute for Future Environments, Queensland University of Technology, Australia
a.sorensen@qut.edu.au

***Abstract*—User interfaces for source code editing are a crucial component in any software development environment, and in many editors visual annotations (overlaid on the textual source code) are used to provide important contextual information to the programmer. This paper focuses on the real-time programming activity of 'cyberphysical' programming, and considers the type of visual annotations which may be helpful in this programming context.**

I. Introduction

In the broadest sense, live programming is *interactive* programming. Commonly this style of programming is associated with some form of read-eval-print loop (REPL) or other top-level interaction vehicle. Interactive programming environments support code interpretation and symbol (re)binding on-the-fly to enable software to be substantially altered during execution. More recently, different terms such as just-in-time programming, livecoding, live programming and cyberphysical programming have been used to extend and clarify the specific technologies and communities which operate under this umbrella.

- *Just-in-time* (JIT) programming, a term initially coined by Richard Potter [1], is "the implementing of algorithms during task-time, the time when the user is actually trying to accomplish the task." This is the broadest definition of live programming, and enfolds all the others in this list.
- *Livecoding* [2] is an audiovisual performance practice where artist-programmers develop generative audiovisual systems live in front of an audience.
- *Live Programming* involves the *direct* construction, manipulation and visualisation of a program's run-time state, including internal structures. These types of environments date back to Self [3], and perhaps the most radical example of this is Jonathon Edwards' SubText [4]. In SubText, run-time program modifications are not standard name (re)bindings (as is the case with interactive REPL-style environments) but require the direct manipulation of a running program's state.
- *Cyberphysical programming* extends the real-time nature of livecoding beyond audiovisual systems[1] to *any* real-time domain, emphasising the relationship between the programmer, the machine, and the environment. This relationship is abstract in nature (i.e. procedural rather than gestural) making the activity one of real-time procedural orchestration. This real-time orchestration may be in the audiovisual domain (as is the case for livecoding), or through some other real-world interaction; controlling a robot or modifying a sensor network. Cyberphysical programming is about building real-time systems *in real-time*, but more fundamentally is about the "reacting responsively to perturbations in the world" [5].

[1]so livecoding is a sub-domain of cyberphysical programming

This paper focuses on cyberphysical programming and asks the question *what information should we give the cyberphysical programmer about the state of the world and its relationship to the code they are editing?* In particular, we consider how visual annotations, overlaid on the textual source code, can assist the cyberphysical programmer. To this end, we present some of our own ongoing work in visual source code annotations in the Impromptu/Extempore [6] cyberphysical programming environment.

II. Visual Annotations: Providing Context to the Programmer

User interfaces for source code editing are a crucial component in a software development environment [7]. In this context, visual annotations are used to provide "relevant, yet passive context" to the programmer [8]. Visual annotations are a near-ubiquitous feature of both commercial and open-source text editors and IDEs.

The visuals annotations may take many different forms. At a very basic level, syntax colouring is the practice of using colour to provide context to the programmer about the syntactic and semantic characteristics of the various tokens in the code [9]. Coloured indicators may also be placed in the 'margins' of the editor [10]. Slightly more sophisticated annotations may include graphical overlays which provide information about code churn and test coverage [8] or provide warnings about code smells [11].

Visual annotations may be continuous (always-on, providing constant feedback about some aspect of the system) or discrete (triggered by certain events or states of the world). There are several factors which influence the attentional draw (AD) of different visual information displays, including colour, size, and movement. While the influence of these factors is complex

978-1-4673-6265-8/13 © 2013 IEEE

LIVE 2013, San Francisco, CA, USA

and context-dependent, in general the best balance of annoyance, perceived benefit, and performance increase occurred when the AD of the notifications matched their criticality (importance) [12].

III. The State of the World and State of the Code

In any programming activity (not just cyberphysical programming), a running program calls procedures (and subprocedures) which change the state of the world, mutating variables and passing data in and out. As such, the *state of the world* (SoW)—the bits in memory, the call stack, readings from sensors and data flow to output devices—is constantly in flux.

The *state of the code* (SoC), in contrast, is the structure of the code independent of its execution. This includes basic problems such as syntax errors, which will prevent the code from even compiling, and also static analysis techniques for determining unused code paths or providing code refactoring suggestions. These analyses are independent of the SoW (indeed the program need not even be running to perform them) but are also important context for the programmer.

What makes cyberphysical programming different from other programming practices is the relationship between these states: the state of the world and the SoC. In particular, the difficulty stems from the fact that the SoW and the SoC can diverge. This matters in cyberphysical programming because the code is *live*—executing even as it is being edited.

An example in Extempore may help to illustrate this point. Consider a quality control robot `qc-bot` which monitors a production line. We want the robot to take a sample from the production line once per second, and if the sample is overweight, to halt production. There are many ways to do this, but one way is with a the 'temporal recursion' design pattern [5]: an asynchronous 'callback' which recursively reschedules itself:

```
(define qc-bot
  (lambda (time)
    ;; take sample from the production line
    (let ((sample (take-sample)))
      ;; if sample is too light, stop everything
      (if (> (weigh-sample sample) 0.5)
          (begin (stop-production)
                 (print "Overweight! Stopping.."))
          ;; else test again in 1 second
          (callback (+ time *second*) 'qc-bot)))))

;; start qc-bot's temporal recursion
(qc-bot (now))
```

Notice that the last thing the `qc-bot` function does before it exits is reschedule itself (through the `callback` function) to be executed at a time one second into the future, creating a temporal 'loop'[2]. When the `qc-bot` function is evaluated (by a key command from the programmer) the temporal recursion is 'kicked off', and will continue to execute, once per second, in perpetuity. In a cyberphysical context, this means that the robot is actually working, taking a sample each second and potentially stopping production if an overweight sample is found.

[2]although it is not a synchronous loop in the `dotimes` sense

Things get even more interesting when the `qc-bot` function is *modified* while it is temporally recursing. Stopping the production line if just one sample is overweight is too drastic—let's edit the code to change the robot's behaviour to a 'three strikes' policy:

```
(define qc-bot
  (let ((strikes 0))
    (lambda (time)
      ;; take sample from the production line
      (let ((sample (take-sample)))
        (if (> (weigh-sample sample) 0.5)
            ;; if 3 strikes, stop production
            (if (>= strikes 3)
                (begin (stop-production)
                       (print "3 strikes! Stopping.."))
                (set! strikes (+ strikes 1)))
            ;; else test again in 1 second
            (callback (+ time *second*) 'qc-bot))))))
```

Until re-evaluation of the `qc-bot` function is triggered by the programmer, the old (one strike) version is still running, and the robot is still behaving in that way. At this point, the source code in the editor and the program being executed have diverged, leaving the editor is in an inconsistent state.

This inconsistency is only possible in live programming environments which require programmer intervention to trigger code compilation and execution. It is not possible in environments like SubText, where there is no notion of 'triggering' evaluation—any manipulations of the interface take effect automatically and immediately in the running program. The tight coupling between code and program is bi-directional, so that changes in the program trigger modifications to the source code.

However, the notion of evaluation as a conscious intervention by the programmer at a controlled moment in time (which is the case in Extempore) has some benefits. Going back to our robot example from earlier, the `qc-bot` checks to see if the weight of the sample is greater than `0.5`. However, to type the literal `0.5` in the usual left-to-right fashion requires three keystrokes, and after the first keystroke only the `0` is present in the editor. If the program was automatically updated in response to any code changes, at that brief moment the inequality check will always return true (assuming that `weigh-sample` returns a positive number). If the robot is checking a sample at that split second, the test will fail and the production line will grind to a halt!

It is therefore desirable in cyberphysical programming to retain manual control over the evaluation of code so that the code can be edited 'through' an incorrect state to get to a new, more desirable one (even if it means that the SoW-SoC divergence is unavoidable). This is a challenge unique to cyberphysical programming—or at least any programming activity which supports the scheduling of a function before it is (re)defined. The programmer needs contextual information about *both* the SoW and SoC to do their job properly, and this is where graphical code annotations can help.

IV. Three States of Visualisation

Because of the primacy of *programming* (editing the source code of a program) in cyberphysical programming, we restrict

ourselves to visual annotations which are overlaid on the code itself (that is, in the text editor window). Harward et. al. [13] call these type of overlays *in-situ* software visualisation.

A. 'State of the World' Annotations

SoW annotations represent tracing information about the state of the *running* program. This information can be obtained through profiling of the running code, or through callback hooks in the program's infrastructure which trigger on certain events. Simply put, SoW annotations are anything that represents information which cannot be determined by looking at the code in isolation from the running program.

SoW annotations may provide continuous information about variable values or may be event-based, such as a notification that a particular piece of code has been executed or a particular condition has been met. The annotations may provide higher-level information than the values of individual variables, too, for example the program's CPU and memory usage. In an audio generation context, knowing the volume and spectral profile of the output audio signal may be desirable. Similarly, in a graphics-based cyberphysical programming context, information about the frame-rate may be of use. *Which* values and functions are annotated in this way depends on the programming context, and to an extent the desires of the programmer.

In our Impromptu heads-up display[3] (HUD), the state of variables such as `sample` (i.e. the clock), as well as the memory and CPU use of the program, can be shown (textually) at the top of the editor window as shown in figure 1. For the cyberphysical programmer these values are fundamental to the scheduling of temporal events, and so it is useful to have their state visible at all times.

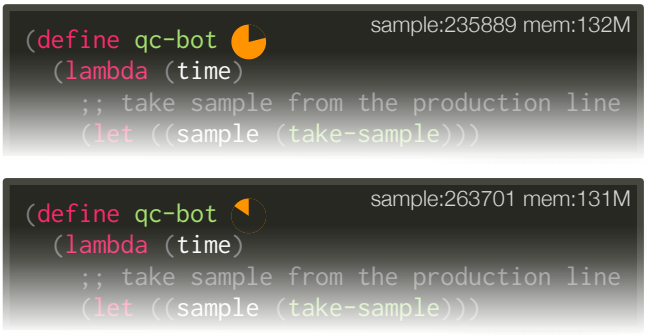

Fig. 1. A clock annotation for the `qc-bot` temporal recursion, which 'spins' at the callback rate (the two panes represent snapshots at successive points in time). Other important state information is shown in the top right-hand corner.

All of these annotations are dynamic and update in real-time, indeed it almost goes without saying that in cyberphysical programming visual annotations must provide real-time feedback, showing the state of the running program *now*. This is more than just a latency issue, though, they can also provide the programmer with information about the temporal patterns of activation in the code—not just *what* is getting executed and with what argument values, but *when* code is being executed.

This is particularly pertinent to the Impromptu/Extempore programmer, who may have multiple temporal recursion loops running simultaneously. In the Impromptu HUD, small 'clocks' are used to indicate the timing of each temporal recursion loop (see figure 1). These clocks support both constant and variable callback rates, and provide visual cues about when each piece of code is being executed. Interacting with the world often involves the precise temporal scheduling of events, and so this temporal dimension of a program's execution is particularly important information for the cyberphysical programmer.

B. 'State of the Code' Annotations

Static analysis techniques (of which there are many) could be of great potential benefit to the cyberphysical programmer, mainly due to the "danger of running very complicated new code live" [2]. If syntax errors and infinite loops[4] can be caught *before* the code goes live (which is what static analysis is all about) then unfortunate crashes can be avoided.

The most basic SoC annotation is syntax colouring, which is near ubiquitous in modern (and even not so modern) source code editors (including Impromptu/Extempore). Techniques such as using specific colours for numeric literals can be helpful when the programmer is scanning the code for interesting values to change.

There are some static analyses which are specific to a temporal-recursion-based program. For example the callback clocks described in the previous section require the detection of these patterns in the code, and in many cases the callback rates of these loops can be determined statically. To continue the `qc-bot` example from earlier, say the head of the factory wanted to use multiple quality control robots on the production line, each with a different sampling rate. They cyberphysical programmer could, while the existing `qc-bot` code was running, edit the function to add `bot` and `rate` arguments. Then, new loops could be triggered with the appropriate arguments, resulting in multiple temporal recursion loops running simultaneously. Information about their scheduling could be provided to the programmer through a scrolling 'piano roll' based overlay as shown in figure 2.

This annotation may change as the code changes, but can still provide helpful context to the programmer for coordinating multiple temporal recursion loops in cyberphysical programming.

C. 'SoC-SoW Relationship' Annotations

The relationship *between* the SoW and the SoC is an area of rich potential for visual annotations. At a very basic level, indicators (e.g. as coloured bars in the margins) could be used to differentiate between code which is running as written and code which has been modified since the last evaluation (and therefore represents a SoW-SoC divergence). This is not

[3]which can be seen in action at `http://vimeo.com/25699729`

[4]Obviously fixing this issue in the most general case involves solving the halting problem, but heuristics can provide protection in specific situations.

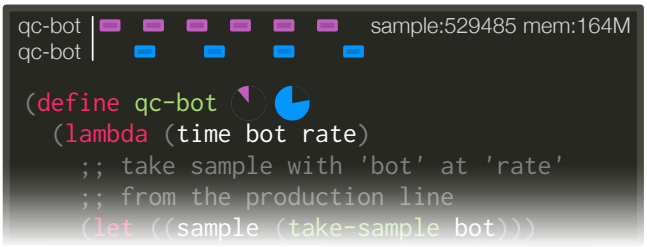

Fig. 2. A piano roll overlay for multiple temporal recursions. There are *two* separate `qc-bot`recursions going on here (indicated with different colours), each with a different `rate` argument. As time goes on, the 'events' scroll to the left until they hit the 'execution line' and are triggered.

dissimilar from the feedback which some editors (e.g. Visual Studio) use to indicate which lines of code have been modified since the last save.

However, it is not only important to know whether the code has diverged from the world, but whether or not the updated code still 'fits' in the contexts where it is already in use. For example, consider an extension to the Impromptu HUD which uses the colour of the spinning clock to indicate whether the modified code (which is involved in a currently running temporal recursion loop) still fits in the running program. This may involve an arity check, or in strongly-typed languages (such as Extempore) a type check.

More useful in a cyberphysical context, though, is a temporal check—an analysis of the temporal appropriateness of a given piece of code (as suggested in [5]). The system could perform analysis of both the running program state to determine overall system load, and also (using heuristics) calculate the approximate cost (in CPU cycles and memory) of the updated code. In high-level languages calculating an upper bound on the execution time of a piece of code is difficult, however in lower-level environments like Extempore, which uses the LLVM compiler infrastructure [14], more accurate estimates (and even guarantees [15]) may be possible.

Visual annotations can then be presented to the programmer regarding how much load the new code will place on the system, and whether this load is manageable. These annotations are much more complex than simple syntax colouring and memory readouts, but they are potentially of much more use to the cyberphysical programmer because they assist in synthesising knowledge about the SoW and the SoC—one of the key challenges in cyberphysical programming.

One final point worth making is that the annotations themselves may be live programmed, or at least live-modifiable. A fundamental tenet of live programming is that there are desirable behaviours which cannot be forseen, and offering the programmer the chance to change the annotations they are presented seems wholly consistent with this idea.

V. Conclusion

There is much work to be done in understanding live programming in general, including in providing visual annotations to assist them in their programming. In this paper we have deliberately limited the discussion to cyberphysical programming, the orchestration (through a textual language) of behaviours by a programmer in time and in the world.

This paper has covered passive annotations out of a desire to keep cyberphysical programming a *programming* (rather than graphical) activity. While some of the annotations discussed in this workshop paper have been implemented in the Impromptu HUD, more work remains to be done both in designing and evaluating the visual annotations presented. Still, an understanding of the complex relationship between the state of the world and the state of the code is necessary to provide maximum benefit to the cyberphysical programmer through visual annotations.

References

[1] R. Potter, "Just-in-time programming, Watch what I do: programming by demonstration," MIT Press, 1993.

[2] A. Mclean, N. Collins, and J. Rohrhuber, "Live coding in laptop performance," *Organised Sound*, vol. 8, no. 3, p. 321, Jan. 2003.

[3] D. Ungar, R. B. Smith, D. Ungar, and R. B. Smith, *Self: The power of simplicity*. ACM, Dec. 1987, vol. 22.

[4] J. Edwards, "No ifs, ands, or buts: Uncovering the simplicity of conditionals," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 639–658, 2007.

[5] A. Sorensen and H. J. Gardner, "Programming with time: cyber-physical programming with impromptu," *OOPSLA '10: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2010.

[6] A. Sorensen. Extempore. [Online]. Available: http://extempore.moso.com.au/

[7] M. L. Van De Vanter, S. L. Graham, and R. A. Ballance, "Coherent user interfaces for language-based editing systems," *Decision Support Systems*, vol. 37, no. 4, pp. 431–466, Oct. 1992.

[8] N. Lopez and A. van der Hoek, "The code orb," in *ICSE '11*. New York, New York, USA: ACM Press, 2011, p. 824.

[9] R. M. Baecker and A. Marcus, *Human factors and typography for more readable programs*. ACM, Nov. 1989.

[10] J. Śliwerski, T. Zimmermann, and A. Zeller, "HATARI: raising risk awareness," in *ESEC/FSE-13*. ACM Request Permissions, Sep. 2005.

[11] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *SOFTVIS '10*. New York, New York, USA: ACM Press, 2010, pp. 5–14.

[12] J. Gluck, A. Bunt, and J. McGrenere, "Matching attentional draw with utility in interruption," in *CHI '07*. New York, New York, USA: ACM Press, 2007, pp. 41–50.

[13] M. Harward, W. Irwin, and N. Churcher, "In Situ Software Visualisation," in *Software Engineering Conference (ASWEC), 2010 21st Australian*, 2010, pp. 171–180.

[14] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[15] F. Merz, S. Falke, and C. Sinz, "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR," in *Verified Software: Theories*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 146–161.